Runtime randomization and perturbation for virtual machines.

JAVIER CABRERA ARTEAGA

Licentiate Thesis in [Research Subject - as it is in your ISP]
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden [2022]

KTH School of Information and
Communication Technology
SE-164 40 Kista
SWEDEN

TRITA-ICT XXXX:XX
ISBN XXX-XX-XXXX-XXX-X

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av licentiatexamen i [ämne/subject] [veckodag/weekday] den [dag/day] [månad/month] [år/2022] klockan [tid/time] i [sal/hall], Electrum, Kungl Tekniska högskolan, Kistagången 16, Kista.

Tryck: Universitetsservice US AB

Abstract

Write your abstract here...
Keywords: Keyword1, keyword2, ...

Sammanfattning

Write your Swedish summary (popular description) here...
Keywords: Keyword1, keyword2, ...

Acknowledgements

Write your professional acknowledgements here...

Acknowledgements are used to thank all persons who have helped in carrying out the research and to the research organizations/institutions and/or companies for funding the research.

Name Surname,
Place, Date

<a href="https://www.flaticon.es/iconos-gratis/personalizado" title="personalizado iconos">Personalizado iconos creados por monkik - Flaticon</a>

<a href="https://www.flaticon.es/iconos-gratis/computadora" title="computadora iconos">Computadora iconos creados por Freepik - Flaticon</a>

Contents

Chapter 1

Background & State of the art

This chapter discusses state of the art in the areas of *WebAssembly* and *Software Diversification*. In Section 1.1 we discuss the WebAssembly language, its motivation, how WebAssembly binaries are generated, language specification, and security-related issues. In Section 1.2, we present a summary of Software Diversification, its foundational concepts and highlighted related works. We select the discussed works by their novelty, critical insights, and representativeness of their techniques. In Section 1.3, we finalize the chapter by stating our novel contributions and comparing them against state-of-the-art related works.

1.1 WebAssembly overview

Over the past decades, JavaScript has been used in the majority of the browser clients to allow client-side scripting. However, due to the complexity of this language and to gain in performance, several approaches appeared, supporting different languages in the browser. For example, Java applets were introduced on web pages late in the 90's, Microsoft made an attempt with ActiveX in 1996 and Adobe added ActionScript later on 1998. All these attempts failed to persist, mainly due to security issues and the lack of consensus on the community of browser vendors.

In 2014, Emscripten proposed with a strict subset of JavaScript, asm.js, to allow low level code such as C to be compiled to JavaScript itself. Asm.js was first implemented as an LLVM backend. This approach came with the benefits of having all the ahead-of-time optimizations from LLVM, gaining in performance on browser clients [32] compared to standard JavaScript code. The main reason why asm.js is faster, is that it limits the language features to those that can be optimized in the LLVM pipeline or those that can be directly translated from the source code. Besides, it removes the majority of the dynamic characteristics of the language, limiting it to numerical types, top-level functions, and one large array in the memory directly accessed as raw data. Since asm.js is a subset of JavaScript it was compatible with all engines at that moment. Asm.js demonstrated that client-code could be improved with the right language design and standarization. The work of Van Es et al. [26] proposed to shrink JavaScript to asm.js in a source-to-source strategy, closing the cycle and extending the fact that asm.js was mainly a compilation target for C/C++ code. Despite encouraging results, JavaScript faces several limitations related to the characteristics of the language. For example, any JavaScript engine requires the parsing and the recompilation of the JavaScript code which implies significant overhead.

Following the asm.js initiative, the W3C publicly announced the WebAssembly (Wasm) language in 2015. WebAssembly is a binary instruction format for a stack-based virtual machine and was officially stated later by the work of Haas et al. [25] in 2017. The announcement of WebAssembly marked the first step of standarizing bytecode in the web environment. Wasm is designed to be fast, portable, self-contained and secure, and it outperforms asm.js [25]. Since 2017, the adoption of WebAssembly keeps growing. For example; Adobe, announced a full online version of Photoshop[1] written in WebAssembly; game companies moved their development from JavaScript to Wasm like is the case of a full Minecraft version [2]; and the case of Blazor [3], a .Net virtual machine implemented in Wasm, able to execute C# code in the browser.

From source to Wasm

All WebAssembly programs are compiled ahead-of-time from source languages. LLVM includes Wasm as a backend since version 8.0.0, supporting a broad range of frontend languages such as C/C++, Rust, Go or AssemblyScript[4]. The resulting binary, works similarly to a traditional shared library, it includes instruction codes, symbols and exported functions. In Figure 1.1, we illustrate the workflow from the creation of Wasm binaries to their execution in the browser. The process starts by compiling the source code program to Wasm (Step ①). This step includes ahead-of-time optimizations. For example, if the Wasm binary is generated out of the LLVM pipeline, all optimizations in the LLVM

The step ② builds the standard library for Wasm usually as JavaScript code. This code includes the external functions that the Wasm binary needs for its execution inside the host engine. For example, the functions to interact with the DOM of the HTML page are imported in the Wasm binary during its call from the JavaScript code. The standard library can be manually written, however, compilers like Emscripten, Rust and Binaryen can generate it automatically, making this process completely transparent to developers.

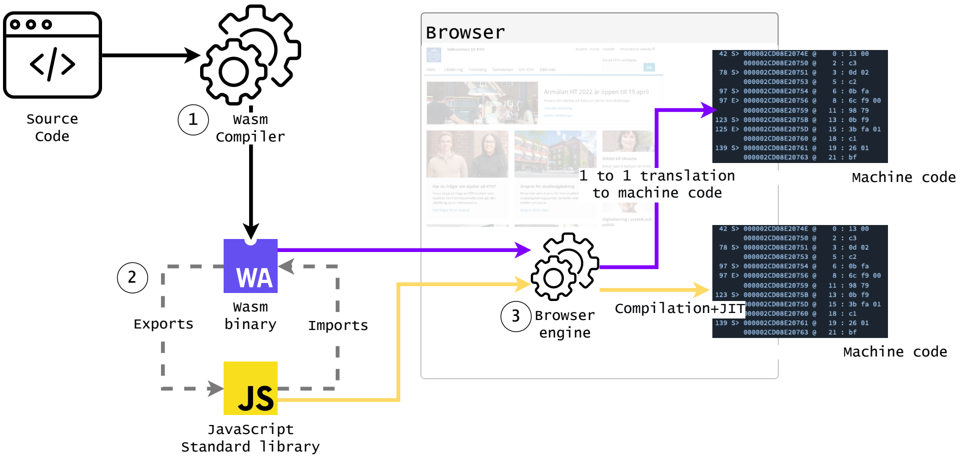
[1] `https://twitter.com/Adobe/status/1453034805004685313?s=20&t=Zf1N7-WmzecA0K4V8R69lw`

[2] `https://satoshinm.github.io/NetCraft/`

[3] `https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor`

[4] subset of the TypeScript language

Finally, the third step (Step ③), includes the compilation and execution of the client-side code. Most of the browser engines compile either the Wasm and JavaScript codes to machine code. In the case of JavaScript, this process involves JIT and hot code replacement during runtime. For Wasm, since it is closer to machine code and it is already optimized, this process is a one-to-one mapping. For instance, in the case of V8, the compilation process only applies simple and fast optimizations such as constant folding and dead code removal. Once V8 completes the compilation process, the generated machine code for Wasm is final and is the same used along all its executions. This analysis was validated by conversations with the V8's dev team and by experimental studies in our previous contributions.



```
 42 S> 000002CD08E2074E @    0 : 13 00
       000002CD08E20750 @    2 : c3
 78 S> 000002CD08E20751 @    3 : 0d 02
       000002CD08E20753 @    5 : c2
 97 S> 000002CD08E20754 @    6 : 0b fa
 97 E> 000002CD08E20756 @    8 : 6c f9 00
       000002CD08E20759 @   11 : 98 79
123 S> 000002CD08E2075B @   13 : 0b f9
125 E> 000002CD08E2075D @   15 : 3b fa 01
       000002CD08E20760 @   18 : c1
139 S> 000002CD08E20761 @   19 : 26 01
       000002CD08E20763 @   21 : bf
```

```
 42 S> 000002CD08E2074E @    0 : 13 00
       000002CD08E20750 @    2 : c3
 78 S> 000002CD08E20751 @    3 : 0d 02
       000002CD08E20753 @    5 : c2
 97 S> 000002CD08E20754 @    6 : 0b fa
 97 E> 000002CD08E20756 @    8 : 6c f9 00
       000002CD08E20759 @   11 : 98 79
123 S> 000002CD08E2075B @   13 : 0b f9
        00002CD08E2075D @   15 : 3b fa 01
        00002CD08E20760 @   18 : c1
139 S> 000002CD08E20761 @   19 : 26 01
       000002CD08E20763 @   21 : bf
```

Figure 1.1: WebAssembly building, compilation in the host engine and execution.

Wasm can execute directly and is platform independent, making it useful for IoT and Edge computing [2, 11]. For instance, Cloudflare and Fastly adapted their platforms to provide Function as a Service (FaaS) directly with WebAssembly. In this case, the standard library, instead of JavaScript, is provided by any other language stack that the host environment supports. In 2019, the Bytecode Alliance [5] proposed WebAssembly System Interface (WASI) [6]. WASI is the foundation to build Wasm code outside of the browser with a POSIX system interface platform. It standarizes the adoption of WebAssembly outside web browsers [13] in heterogeneous platforms.

WebAssembly specificities

WebAssembly defines its own Instruction Set Architecture (ISA) [22]. It is an abstraction close to machine code instructions but agnostic to CPU architectures. Thus, Wasm is platform independent. The ISA of Wasm includes also the necessary

[5] `https://bytecodealliance.org/`

components that the binary requires to run in any host engine. A Wasm binary has a unique module as its main component. A module is composed by sections, corresponding to 13 types, each of them with an explicit semantic and a specific order inside the module. This makes the compilation to machine code faster.

In Listing 1.1 and Listing 1.2 we illustrate a C program and its compilation to Wasm. The C function contains: heap allocation, external function declaration and the definition of a function with a loop, conditional branching, function calls and memory accesses. The code in Listing 1.2 is in the textual format for the generated Wasm. The module in this case first defines the signature of the functions(Line 2, Line 3 and Line 4) that help in the validation of the binary defining its parameter and result types. The information exchange between the host and the Wasm binary might be in two ways, exporting and importing functions, memory and globals to and from the host engine (Line 5, Line 35 and Line 36). The definition of the function (Line 6) and its body follows the last import declaration at Line 5.

The function body is composed by local variable declarations and typed instructions that are evaluated in a virtual stack (Line 7 to Line 32 in Listing 1.2). Each instruction reads its operands from the stack and pushes back the result. The result of a function call is the top value of the stack at the end of the execution. In the case of Listing 1.2, the result value of the main function is the calculation of the last instruction, `i32.add` at Line 32. A valid Wasm binary should have a valid stack structure that is verified during its translation to machine code. The stack validation is carried out using the static types of Wasm, `i32`, `i64`, `f32` and `f64`. As the listing shows, instructions are annotated with a numeric type.

Wasm manages the memory in a restricted way.A Wasm module has a linear memory component that is accessed as `i32` pointers and should be isolated from the virtual stack. The declaration of the linear data in the memory is showed in Line 37. The memory access is illustrated in Line 15. This memory is usually bound in browser engines to 2Gb of size, and it is only shareable between the process that instantiate the Wasm binary and the binary itself (explicitly declared in Line 33 and Line 36). Therefore, this ensures the isolation of the execution of Wasm code.

Wasm also provides global variables in their four primitive types. Global variables (Line 34) are only accessible by their declaration index, and it is not possible to dynamically address them. For functions, Wasm follows the same mechanism, either the functions are called by their index (Line 30) or using a static table of function declarations. This latter allows modeling dynamic calls of functions (through pointers) from languages such as C/C++; however, the compiler should populate the static table of functions.

In Wasm, all instructions are grouped into blocks, being the start of a function the root block. Two consecutive block declarations can be appreciated in Line 10 and Line 11 of Listing 1.2. Control flow structures jump between block boundaries and not to any position in the code like regular assembly code. A block may specify the state that the stack must have before its execution and the result stack value coming from its instructions. Inside the Wasm binary the blocks explicitly define where they start and end (Line 25 and Line 28). By design, each block executes

Listing 1.1: Example C function.

```
// Some raw data
const int A[250];

// Imported function
int ftoi(float a);

int main() {
    for(int i = 0; i < 250; i++) {
        if (A[i] > 100)
            return A[i] + ftoi(12.54);
    }

    return A[0];
}
```

Listing 1.2: WebAssembly code for Listing 1.1.

```
 1 (module
 2   (type (;0;) (func (param f32) (result i32)))
 3   (type (;1;) (func))
 4   (type (;2;) (func (result i32)))
 5   (import "env" "ftoi" (func $ftoi (type 0)))
 6   (func $main (type 2) (result i32)
 7     (local i32 i32)
 8     i32.const -1000
 9     local.set 0 ;loop iteration counter;
10     block  ;label = @1;
11       loop  ;label = @2;
12         i32.const 0
13         local.get 0
14         i32.add
15         i32.load
16         local.tee 1
17         i32.const 101
18         i32.ge_s ;loop iteration condition;
19         br_if 1 ;@1;
20         local.get 0
21         i32.const 4
22         i32.add
23         local.tee 0
24         br_if 0 ;@2;
25       end
26       i32.const 0
27       return
28     end
29     f32.const 0x1.9147aep+3 ;=12.54;
30     call $ftoi
31     local.get 1
32     i32.add)
33   (memory (;0;) 1)
34   (global (;4;) i32 (i32.const 1000))
35   (export "memory" (memory 0))
36   (export "A" (global 2))
37   (data $data (0) "\00\00\00\00...")
38 )
```

independently and cannot execute or refer to outer block values. This is quarantieed by explicitly annotating the state of the stack before and after the block. Three instructions handle the navigation between blocks: unconditional break, conditional break (Line 19 and Line 24) and table break. Each break instruction can only jump to one of its enclosing blocks. For example, in Listing 1.2, Line 19 forces the execution to jump to the end of the first block at Line 10 if the value at the top of the stack is greater than zero.

We want to remark that de description of Wasm in this section follows the version 1.0 of the language and not its proposals for extended features. We follow those features implemented in the majority of the vendors according to the Wasm roadmap [23]. On the other hand we excluded instructions for datatype conversion, table accesses and the majority of the arithmetic instructions for the sake of simplicity.

WebAssembly's security

As we described, WebAssembly is deterministic and well-typed, follows a structured control flow and explicitly separates its linear memory model, global variables and the execution stack. This design is robust [12] and makes easy for compilers and engines to sandbox the execution of Wasm binaries. Following the specification of Wasm for typing, memory, virtual stack and function calling, host environments should provide protection against data corruption, code injection, and return-oriented programming (ROP).

However, WebAssembly is vulnerable under certain conditions, at the execution engine's level [17]. Implementations in both browsers and standalone runtimes [2] are vulnerable. Genkin et al. demonstrated that Wasm could be used to exfiltrate data using cache timing-side channels [20]. One of our previous contributions trigger a CVE[6] on the code generation component of wasmtime, highlighting that even when the language specification is meant to be secure, the underlying host implementation might not be. Moreover, binaries itself can be vulnerable. The work of Lehmann et al. [9] proved that C/C++ source code vulnerabilities can propagate to Wasm such as overwriting constant data or manipulating the heap using stackoverflow. Even though these vulnerabilities need a specific standard library implementation to be exploited, they make a call for better defenses for WebAssembly. Several proposals for extending WebAssembly in the current roadmap could address some existing vulnerabilities. For example, having multiple memories could incorporate more than one memory, stack and global spaces, shrinking the attack surface. However, the implementation, adoption and settlement of the proposals are far from being a reality in all browser vendors.

1.2 Software Diversification

Software Diversification has been widely studied in the past decades. This section discusses its state of the art. Software diversification consists in synthesizing, reusing, distributing, and executing different, functionally equivalent programs. According to the survey of Baudry and Monperrus [31], the motivation for software diversification can be separated in five categories: reusability [50], software testing [42], performance [39], fault tolerance [59] and security [57]. Our work contributes to the latter two categories. In this section we discuss related works by highlighting how they generate diversification and how they use the generated diversification.

There are two primary sources of software diversification: Natural and Artificial Diversity [31]. This work contributes to the state of the art of Artificial Diversity, which consists of artificially synthesizing software. We have found that the foundation for artificial software diversity has barely changed since Cohen in 1993 [57]. Therefore, the work of Cohen is the cornerstone of this dissertation.

[6] `https://www.fastly.com/blog/defense-in-depth-stopping-a-wasm-compiler-bug-before-it-became-a-problem`

Variants' generation

Cohen et al. proposed to generate artificial software diversification through mutation strategies. A mutation strategy is a set of rules to define how a specific component of software development should be changed to provide a different yet functionally equivalent program. Cohen et al. proposed 10 concrete transformation strategies that can be applied at fine-grained level. We summarize them, complemented with the work of Baudry and Monperrus [31] and the work of Jackson et al. [37], in 5 strategies.

(S1) *Equivalent instructions replacement* Semantically equivalent code can replace pieces of programs. For example, it replaces the original code with equivalent arithmetic expressions or injects instructions that do not affect the computation result(garbage instructions). There are two main approaches for generating equivalent code: rewriting rules and exhaustive searching. The replacement strategies are written by hand as rewriting rules for the first one. A rewriting rule is a tuple composed of a piece of code and a semantic equivalent replacement. For example, Cleemput et al. [38] and Homescu et al. [35] introduce the usage of inserting NOP instructions to generate statically different variants. In their works, the rewriting rule is defined as `instr => (nop instr)`, meaning that `nop` operation followed by the instruction is a valid replacement . On the other hand, exhaustive searching samples or constructs all possible programs for a specific language. Then, a theorem solver checks each found replacement for semantic equivalence. In this topic, Jacob et al. [44] proposed the technique called superdiversification for x86 binaries. Similarly, Tsoupidi et al. [8] introduced Diversity by Construction, a constraint-based compiler to generate software diversity for MIPS32 architecture.

(S2) *Instruction reordering* This strategy reorders instructions or entire program blocks if they are independent. The location of variable declarations might change as well if compilers resort them in the symbol tables. It prevents static examination and analysis of parameters and alters memory locations. In this field, Bhatkar et al. [54, 51] proposed the random permutation of the order of variables and routines for ELF binaries.

(S3) *Adding, changing, removing jumps and calls* This strategy creates program variants by adding, changing, or removing jumps and calls in the original program. Cohen [57] mainly illustrated the case by inserting bogus jumps in programs. Pettis and Hansen [58] proposed to split basic blocks and functions for the PA-RISC architecture, inserting jumps between splits. Similarly, Crane et al. [30] de-inline basic blocks of code as an LLVM pass. In their approach, each de-inlined code is transformed into semantically equivalent functions that are randomly selected at runtime to replace the original code calculation. On the same topic, Bhatkar et al. [51] extended their previous approach [54], replacing function calls by indirect pointer calls in C source code, allowing post binary reordering of function calls. Recently, Romano et al. [1] proposed an obfuscation technique for JavaScript in which part of the code is replaced by calls to complementary Wasm function.

(S4) *Program memory and stack randomization* This strategy changes the layout of programs in the host memory. Also, it can randomize how a program variant operates its memory. The work of Bhatkar et al. [54, 51] also proposed to randomize the base addresses of applications and the library memory regions, and the random introduction of gaps between memory objects in ELF binaries. Tadesse Aga and Autin [16] and Lee et al. [3] recently proposed a technique to randomize the local stack organization for function calls using a custom LLVM compiler. Younan et al. [48] proposed to separate a conventional stack into multiple stacks where each stack contains a particular class of data. On the same topic, Xu et al. [7] transform programs to reduce memory exposure time, improving the time needed for frequent memory address randomization.

(S5) *ISA randomization and simulation* This strategy encodes the original program binary. Once encoded, the program can be decoded only once at the target client, or it can be interpreted in the encoded form using a custom virtual machine implementation. This technique is strong against attacks involving the examination of code. Kc et al. [53] and Barrantes et al. [55] proposed seminal works on instruction-set randomization to create a unique mapping between artificial CPU instructions and real ones. On the same topic, Chew and Song [56] target operating system randomization. They randomize the interface between the operating system and the user applications. Couroussé et al. [27] implement an assembly-like DSL to generate equivalent code at runtime in order to increase protection against side-channel attacks. Their technique generates a different program during execution using an interpreter for their DSL.

All described strategies can be mixed together. They can be applied in any sequence and recursively, providing a richer diversity environment. The main limitation is that at some point the code of the variant cannot be differentiated by the mutation strategy. However, this is the same issues that want to create for potential attackers.

Variants' equivalence

Equivalence checking between program variants is an essential component for any program transformation task, from checking compiler optimizations [34] to the artificial synthesis of programs discussed in this chapter. Equivalence checking proves that two pieces of code or programs are semantically equivalent. This process is undecidable; still, some approaches approximate it [15]. Cohen [57] simplifies this process by enunciating the following property: two programs are variants of the same program if given identical input, they produce the identical output. We use this same enunciation as the definition of *semantic/functional equivalence* along with this dissertation.

Notice that this property is relaxed. Two programs can be statically different but still provide the same semantic output. For example, the work of Sengupta et al. [24] uses the rotation of different database engines for reliability. In this case, all database engines provide the same input/output property for creating, updating, and adding new data. Their solution has input/output equivalent programs provided by statically different binaries with different observable behaviors during runtime. On the contrary, the best example of restricted equivalence checking is the one followed by most antivirus applications. In practice, most of them check for the presence of the same binary in an enormous database, *i.e.,* if two programs are exactly the same binary, they are also semantically equivalent.

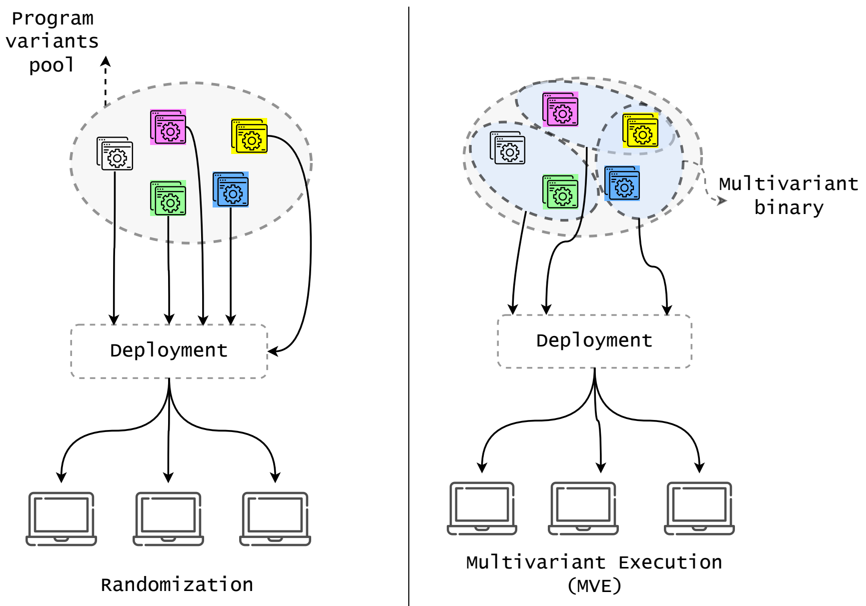
The easiest way to guarantee the equivalence property is by construction. For example, in the case illustrated in S1 for Cleemput et al. [38] and Homescu et al. [35], the transformation strategy is constructed to be semantically equivalent since the beginning. However, this process is prone to developer errors, and better checking is needed. For example, the test suite of the original program can be used to check the variant. If the test suite passes for the program variant [10], it is equivalent to the original. However, it is limited due to the need for a preexisting test suite. When program variants are artificially created, another technique is needed to check for equivalence.

The majority of the previously mentioned works use theorem solvers (SMT solvers) [45] or fuzzers [?] to prove equivalence. For SMT solvers, the main idea is to turn the two codes into math expressions. The SMT solver checks for counter-examples. When the SMT solver finds a counter-example, there exists an input for which the two math expressions return a different output. The main limitation of this technique is that all algorithms cannot be translated to a math expression, for example, loops. Yet, this technique tends to be the most used for no-branching-programs checking like basic block and peephole replacements [?] . On the other hand, fuzzers look randomly for inputs that provide different observable behavior. If two inputs provide a different output in the variant, the variant and the original program are not semantically equivalent. The main limitation for fuzzers is that the process is remarkably time-expensive and requires the introduction of oracles by hand. In practice, a fuzzer needs a significant amount of time to check two programs, and this time takes hours at the minimum.

We have found that SMT solvers are the best option for checking most of the fine-grained transformations previously mentioned for two main reasons. First, the field of SMT is mature and provides battle-tested tooling [45, 19]. Second, the existing SMT tooling is configurable and flexible, making the checking of program variants feasible in terms of the SMT solver execution time. In **??** we describe how we apply SMT solvers in our contributions for equivalence checking.

Usages of Software Diversity

After program variants are generated, they can be used in two main scenarios: Randomization or Multivariant Execution(MVE) [37]. In Figure 1.2a and Figure 1.2b we illustrate both scenarios.



(a) Randomization scenario. Given a pool of program variants, one variant is deployed per host. Each deployment randomly selects which variant is assigned to each host. The same program variant is executed in the host at every program invocation between deployments.

(b) Multivariant Execution scenario. Given a pool of program variants, a sample of the pool is packaged in a multivariant binary that is deployed per. Each deployment randomly selects which multivariant binary is assigned to each host. A variant from the multivariant binary is randomly executed at runtime in the host, .

Figure 1.2: Software Diversification usages.

(U1) *Randomization:* In the first scenario Figure 1.2a, a program is selected from the collection of variants (program's variant pool), and at each deployment, it is assigned to a random client. Jackson et al. [37] call this a herd immunity scenario since vulnerable binaries can affect only part of the client's community. For instance, the Polyverse company [7] uses this scenario in real life. They deliver a unique Linux distribution compilation for each of its clients by scrambling the Linux packages at the source code level.

(U2) *Multivariant Execution(MVE):* In the second scenario Figure 1.2b, multiple program variants are composed in one single binary (multivariant binary). The University of Virginia laid the foundations of this approach [49]. Each multivariant binary is randomly deployed to a client. Once in the client, the multivariant

[7] `https://polyverse.com/`

binary executes its embedded program variants at runtime. The execution of the embedded variants can be either in parallel to check for inconsistencies or a single program to randomize execution paths [54]. Bruschi et al. [47] extended the idea of executing two variants in parallel with not-overlapping and randomized memory layouts. Simultaneously, Salamat et al. [46] modified a standard library that generates 32-bit Intel variants where the stack grows in the opposite direction, checking for memory inconsistencies. Notably, Davi et al. proposed Isomeron [29], an approach for execution-path randomization. Isomeron simultaneously loads the original program and a variant. While the program is running, Isomeron continuously flips a coin to decide which copy of the program should be executed next at the level of function calls. Neatly exploiting the limit case of executing only two variants in a multivariant binary has demonstrated to harden systems [43, 36, 28, 18, 29]. Further two variants per multivariant binary, Agosta et al. [33] and Crane et al. [30] used more than two generated programs in the multivariant composition, randomizing software control flow at runtime.

Both scenarios have demonstrated to harden security by tackling know vulnerabilities such as (JIT)ROP attacks [40] and power side-channels [41]. Moreover, Artificial Software Diversification is a preemptive technique for yet unknown vulnerabilities [37]. In Table 1.1 we annotate the related works with the defined usage scenarios. Our work contributes to both scenarios as a preemptive technique for hardening WebAssembly.

1.3 Statement of Novelty

We contribute to Software Diversification for WebAssembly using Artificial Diversification, for Randomization and Multivariant Execution usages (U1, U2). In Table 1.1 we listed related work on Artificial Software Diversification that support our work. The table is composed by the authors and the reference to their work, followed by one column for each strategy and usage (S1, S2, S3, S4, S5, U1 and U2). The last column of the table summarize their technical contribution and the reach of their work. Each cell in the table contains a checkmark if the strategy or the usage of the work match the previously mentioned classifications. The rows are sorted by the year of the work in ascending order. The last two rows locate our contributions.

Our first contribution, CROW [5] generates multiple program variants for WebAssembly using the LLVM pipeline. It contributes to state of the art in artificially creating randomization for WebAssembly (U1). Because of the specificities of code execution in the browser (mentioned in Section 1.1), this can be considered a randomization approach. For example, since WebAssembly is served at each page refreshment, every time a user asks for a WebAssembly binary, she can be served a different variant provided by CROW. With MEWE [4], our second contribution, we randomly select from several variants at runtime, creating a multivariant execution scheme(U2) that randomizes the observable behaviors at each run of the multivariant binary.

Conclusions

In this chapter, we presented the background on the WebAssembly language, including its security issues and related work. This chapter aims to settle down the foundation to study automatic diversification for WebAssembly. We highlighted related work on Artificial Software Diversification, showing that it has been widely researched, not being the case for WebAssembly. On the other hand, current available implementations for Software Diversification cannot be directly ported to Wasm. We placed our contributions in the field of artificial diversity. In **??** we describe the technical details that lead our contributions. Besides, the impact of our contributions is evaluated by following the methodology described in **??**.

Authors	**S1**	**S2**	**S3**	**S4**	**S5**	**U1**	**U2**	**Main technical contribution**
Pettis and Hansen [58]		✓		✓		✓		Custom Pascal compiler for PA-RISC architecture
Chew and Song [56]			✓			✓		Linux Kernel recompilation.
Kc et al. [53]					✓			Linux Kernel recompilation.
Barrantes et al. [55]					✓	✓		x86 to x86 transformations using Valgrind
Bhatkar et al. [54]	✓	✓		✓		✓		ELF binary transformations
El-Khalil and Keromytis [52]						✓		custom GCC compiler for x86 architecture
Bhatkar et al. [51]	✓	✓		✓		✓		C/C++ source to source transformations and ELF binary transformations
Younan et al. [48]				✓				custom GCC compiler
Bruschi et al. [47]				✓		✓		ELF binary transformations.
Salamat et al. [46]			✓				✓	Custom GNU compiler
Jacob et al. [44]	✓	✓						x86 to x86 transformations
Salamat et al. [43]				✓			✓	x86 to x86 transformations
Amarilli et al. [41]	✓				✓	✓		Polymorphic code generator for ARM architecture
Jackson [37]	✓					✓	✓	LLVM compiler, only backend for x86 architecture
Cleemput et al. [52]	✓					✓		x86 to x86 transformations
Homescu et al. [35]	✓					✓		LLVM 3.1.0†
Crane et al. [30]	✓	✓	✓				✓	LLVM, only backend for x86 architecture
Davi et al. [29]						✓		Windows DLL instrumentation
Couroussé et al. [27]	✓	✓			✓	✓		Custom GCC compiler targeting microcontrollers
Lu et al. [18]				✓			✓	GNU assembler for Linux kernel
Belleville et al. [21]	✓			✓		✓		Only C language frontend, LLVM 3.8.0†
Aga et al. [16]				✓		✓		Data layout randomization, LLVM 3.9†
Österlund et al. [14]				✓			✓	Linux Kernel recompilation.
Xu et al. [7]				✓		✓		Custom kernel module in Linux OS
Lee et al. [3]				✓		✓		LLVM 12.0.0 backend for x86
Cabrera Arteaga et al. [5]	✓	✓	✓	✓		✓		Any frontend language for LLVM version 12.0.0 targeting Wasm backend
Cabrera Arteaga et al. [4]	✓	✓	✓	✓			✓	Any frontend and backend language for LLVM version 12.0.0

† Notice that LLVM only supports WebAssembly backend from version 8.0.0

Table 1.1: The table is composed by the authors and the reference to their work, followed by one column for each strategy and usage (S1, S2, S3, S4, S5, U1 and U2). The last column of the table summarize their technical contribution. Each cell in the table contains a checkmark if the strategy or the usage of the work match the previously mentioned classifications. The rows are sorted by the year of the work in ascending order. The last two rows locate our contributions.

Bibliography

[1] Romano,A., Lehmann,D., Pradel,M., and Wang,W. (2022). Wobfuscator: Obfuscating javascript malware via opportunistic translation to webassembly. In *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*, pages 1101–1116, Los Alamitos, CA, USA. IEEE Computer Society.

[2] Narayan,S., Disselkoen,C., Moghimi,D., Cauligi,S., Johnson,E., Gang,Z., Vahldiek-Oberwagner,A., Sahita,R., Shacham,H., Tullsen,D., et al. (2021). Swivel: Hardening webassembly against spectre. In *USENIX Security Symposium*.

[3] Lee,S., Kang,H., Jang,J., and Kang,B. B. (2021). Savior: Thwarting stack-based memory safety violations by randomizing stack layout. *IEEE Transactions on Dependable and Secure Computing*.

[4] Cabrera Arteaga,J., Laperdrix,P., Monperrus,M., and Baudry,B. (2021). Multi-Variant Execution at the Edge. *arXiv e-prints*, page arXiv:2108.08125.

[5] Cabrera Arteaga,J., Floros,O., Vera Perez,O., Baudry,B., and Monperrus,M. (2021). Crow: code diversification for webassembly. In *MADWeb, NDSS 2021*.

[6] (2021). Webassembly system interface.

[7] Xu,Y., Solihin,Y., and Shen,X. (2020). Merr: Improving security of persistent memory objects via efficient memory exposure reduction and randomization. In *Proc. of ASPLOS*, pages 987–1000.

[8] Tsoupidi,R. M., Lozano,R. C., and Baudry,B. (2020). Constraint-based software diversification for efficient mitigation of code-reuse attacks. *ArXiv*, abs/2007.08955.

[9] Lehmann,D., Kinder,J., and Pradel,M. (2020). Everything old is new again: Binary security of webassembly. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association.

[10] Harrand,N., Soto-Valero,C., Monperrus,M., and Baudry,B. (2020). Java decompiler diversity and its application to meta-decompilation. *Journal of Systems and Software*, 168:110645.

[11] Gadepalli,P. K., McBride,S., Peach,G., Cherkasova,L., and Parmer,G. (2020). Sledge: A serverless-first, light-weight wasm runtime for the edge. In *Proceedings of the 21st International Middleware Conference*, page 265–279.

[12] Chen,D. and W3C group (2020). WebAssembly documentation: Security. Accessed: 18 June 2020.

[13] Bryant,D. (2020). Webassembly outside the browser: A new foundation for pervasive computing. In *Proc. of ICWE 2020*, pages 9–12.

[14] Österlund,S., Koning,K., Olivier,P., Barbalace,A., Bos,H., and Giuffrida,C. (2019). kmvx: Detecting kernel information leaks with multi-variant execution. In *ASPLOS*.

[15] Churchill,B., Padon,O., Sharma,R., and Aiken,A. (2019). Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 1027–1040, New York, NY, USA. Association for Computing Machinery.

[16] Aga,M. T. and Austin,T. (2019). Smokestack: thwarting dop attacks with runtime stack layout randomization. In *Proc. of CGO*, pages 26–36.

[17] Silvanovich,N. (2018). The problems and promise of webassembly. Technical report.

[18] Lu,K., Xu,M., Song,C., Kim,T., and Lee,W. (2018). Stopping memory disclosures via diversification and replicated execution. *IEEE Transactions on Dependable and Secure Computing*.

[19] Gupta,S., Saxena,A., Mahajan,A., and Bansal,S. (2018). Effective use of smt solvers for program equivalence checking through invariant-sketching and query-decomposition. In Beyersdorff,O. and Wintersteiger,C. M., editors, *Theory and Applications of Satisfiability Testing – SAT 2018*, pages 365–382, Cham. Springer International Publishing.

[20] Genkin,D., Pachmanov,L., Tromer,E., and Yarom,Y. (2018). Drive-by key-extraction cache attacks from portable code. *IACR Cryptol. ePrint Arch.*, 2018:119.

[21] Belleville,N., Couroussé,D., Heydemann,K., and Charles,H.-P. (2018). Automated software protection for the masses against side-channel attacks. *ACM Trans. Archit. Code Optim.*, 15(4).

[22] WebAssembly Community Group (2017b). WebAssembly Specification.

[23] WebAssembly Community Group (2017a). WebAssembly Roadmap.

[24] Sengupta,S., Vadlamudi,S. G., Kambhampati,S., Doupé,A., Zhao,Z., Taguinod,M., and Ahn,G.-J. (2017). A game theoretic approach to strategy generation for moving target defense in web applications. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, page 178–186.

[25] Haas,A., Rossberg,A., Schuff,D. L., Schuff,D. L., Titzer,B. L., Holman,M., Gohman,D., Wagner,L., Zakai,A., and Bastien,J. F. (2017). Bringing the web up to speed with webassembly. *PLDI*.

[26] Van Es,N., Nicolay,J., Stievenart,Q., D'Hondt,T., and De Roover,C. (2016). A performant scheme interpreter in asm.js. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, page 1944–1951, New York, NY, USA. Association for Computing Machinery.

[27] Couroussé,D., Barry,T., Robisson,B., Jaillon,P., Potin,O., and Lanet,J.-L. (2016). Runtime code polymorphism as a protection against side channel attacks. In *IFIP International Conference on Information Security Theory and Practice*, pages 136–152. Springer.

[28] Kim,D., Kwon,Y., Sumner,W. N., Zhang,X., and Xu,D. (2015). Dual execution for on the fly fine grained execution comparison. *SIGPLAN Not.*

[29] Davi,L., Liebchen,C., Sadeghi,A.-R., Snow,K. Z., and Monrose,F. (2015). Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *NDSS*.

[30] Crane,S., Homescu,A., Brunthaler,S., Larsen,P., and Franz,M. (2015). Thwarting cache side-channel attacks through dynamic software diversity. In *NDSS*, pages 8–11.

[31] Baudry,B. and Monperrus,M. (2015). The multiple facets of software diversity: Recent developments in year 2000 and beyond. *ACM Comput. Surv.*, 48(1).

[32] Alon Zakai (2015). asm.js Speedups Everywhere.

[33] Agosta,G., Barenghi,A., Pelosi,G., and Scandale,M. (2015). The MEET approach: Securing cryptographic embedded software against side channel attacks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(8):1320–1333.

[34] Le,V., Afshari,M., and Su,Z. (2014). Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 216–226.

[35] Homescu,A., Neisius,S., Larsen,P., Brunthaler,S., and Franz,M. (2013). Profile-guided automated software diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–11. IEEE.

[36] Maurer,M. and Brumley,D. (2012). Tachyon: Tandem execution for efficient live patch testing. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 617–630.

[37] Jackson,T. (2012). *On the Design, Implications, and Effects of Implementing Software Diversity for Security*. PhD thesis, University of California, Irvine.

[38] Cleemput,J. V., Coppens,B., and De Sutter,B. (2012). Compiler mitigations for time attacks on modern x86 processors. *ACM Trans. Archit. Code Optim.*, 8(4).

[39] Sidiroglou-Douskos,S., Misailovic,S., Hoffmann,H., and Rinard,M. (2011). Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, page 124–134, New York, NY, USA. Association for Computing Machinery.

[40] Jackson,T., Salamat,B., Homescu,A., Manivannan,K., Wagner,G., Gal,A., Brunthaler,S., Wimmer,C., and Franz,M. (2011). Compiler-generated software diversity. In *Moving Target Defense*, pages 77–98. Springer.

[41] Amarilli,A., Müller,S., Naccache,D., Page,D., Rauzy,P., and Tunstall,M. (2011). Can code polymorphism limit information leakage? In *IFIP International Workshop on Information Security Theory and Practices*, pages 1–21. Springer.

[42] Chen,T. Y., Kuo,F.-C., Merkel,R. G., and Tse,T. H. (2010). Adaptive random testing: The art of test case diversity. *J. Syst. Softw.*, 83:60–66.

[43] Salamat,B., Jackson,T., Gal,A., and Franz,M. (2009). Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 33–46.

[44] Jacob,M., Jakubowski,M. H., Naldurg,P., Saw,C. W. N., and Venkatesan,R. (2008). The superdiversifier: Peephole individualization for software protection. In *International Workshop on Security*, pages 100–120. Springer.

[45] de Moura,L. and Bjørner,N. (2008). Z3: An efficient smt solver. In Ramakrishnan,C. R. and Rehof,J., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg. Springer Berlin Heidelberg.

[46] Salamat,B., Gal,A., Jackson,T., Manivannan,K., Wagner,G., and Franz,M. (2007). Stopping buffer overflow attacks at run-time: Simultaneous multi-variant program execution on a multicore processor. Technical report, Technical Report 07-13, School of Information and Computer Sciences, UCIrvine.

[47] Bruschi,D., Cavallaro,L., and Lanzi,A. (2007). Diversified process replicæ for defeating memory error exploits. In *Proc. of the Int. Performance, Computing, and Communications Conference*.

[48] Younan,Y., Pozza,D., Piessens,F., and Joosen,W. (2006). Extended protection against stack smashing attacks without performance loss. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 429–438.

[49] Cox,B., Evans,D., Filipi,A., Rowanhill,J., Hu,W., Davidson,J., Knight,J., Nguyen-Tuong,A., and Hiser,J. (2006). N-variant systems: a secretless framework for security through diversity. In *Proc. of USENIX Security Symposium*, USENIX-SS'06.

[50] Pohl,K., Böckle,G., and Van Der Linden,F. (2005). *Software product line engineering: foundations, principles, and techniques*, volume 1. Springer.

[51] Bhatkar,S., Sekar,R., and DuVarney,D. C. (2005). Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the USENIX Security Symposium*, pages 271–286.

[52] El-Khalil,R. and Keromytis,A. D. (2004). Hydan: Hiding information in program binaries. In Lopez,J., Qing,S., and Okamoto,E., editors, *Information and Communications Security*, pages 187–199, Berlin, Heidelberg. Springer Berlin Heidelberg.

[53] Kc,G. S., Keromytis,A. D., and Prevelakis,V. (2003). Countering code-injection attacks with instruction-set randomization. In *Proc. of CCS*, pages 272–280.

[54] Bhatkar,S., DuVarney,D. C., and Sekar,R. (2003). Address obfuscation: an efficient approach to combat a board range of memory error exploits. In *Proceedings of the USENIX Security Symposium*.

[55] Barrantes,E. G., Ackley,D. H., Forrest,S., Palmer,T. S., Stefanovic,D., and Zovi,D. D. (2003). Randomized instruction set emulation to disrupt binary code injection attacks. In *Proc. CCS*, pages 281–289.

[56] Chew,M. and Song,D. (2002). Mitigating buffer overflows by operating system randomization. Technical Report CS-02-197, Carnegie Mellon University.

[57] Cohen,F. B. (1993). Operating system protection through program evolution. *Computers & Security*, 12(6):565–584.

[58] Pettis,K. and Hansen,R. C. (1990). Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, page 16–27, New York, NY, USA. Association for Computing Machinery.

[59] Avizienis and Kelly (1984). Fault tolerance by design diversity: Concepts and experiments. *Computer*, 17(8):67–80.